Classical Concepts in Quantum Programming

Bernhard Ömer

Institute for Theoretical Physics
Technical University Vienna, Austria
oemer@tph.tuwien.ac.at

November 11, 2002

Abstract

The rapid progress of computer technology has been accompanied by a corresponding evolution of software development, from hardwired components and binary machine code to high level programming languages, which allowed to master the increasing hardware complexity and fully exploit its potential.

This paper investigates, how classical concepts like hardware abstraction, hierarchical programs, data types, memory management, flow of control and structured programming can be used in quantum computing. The experimental language QCL will be introduced as an example, how elements like irreversible functions, local variables and conditional branching, which have no direct quantum counterparts, can be implemented, and how non-classical features like the reversibility of unitary transformation or the non-observability of quantum states can be accounted for within the framework of a procedural programming language.

1 Quantum Programming

1.1 Quantum Programming Languages

From a software engineering point of view, we can regard the formalism of Hilbert-space algebra as a specification language, as the mathematical description of a quantum algorithm is inherently declarative and provides no means to derive a unique decomposition into elementary operations for a given quantum hardware.

Low level formalisms like quantum circuits [4], on the other hand, are usually restricted to specific tasks, such as the description of unitary transformations, and thus lack the generality to express all aspects of non-classical algorithms.

The purpose of programming languages is therefore twofold, as they allow to express the semantics of the computation in an abstract manner, as well as the automated generation of a sequence of elementary operations to control the computing device.

1.2 Quantum Algorithms

In its simplest form, a quantum algorithm merely consists of a unitary transformation and a subsequent measurement of the resulting state. For more "traditional" computational tasks, however, as e.g. searching or mathematical calculations, efficient quantum implementations often have the form of probabilistic algorithms. Figure 1 shows the basic outline of a probabilistic non-classical algorithm with a simple evaluation loop.

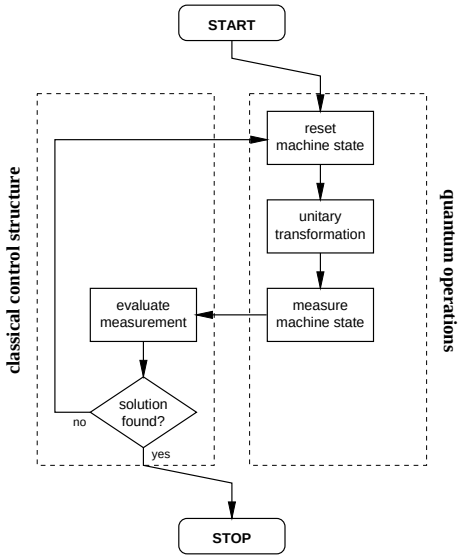

Figure 1: *a simple non-classical algorithm*

More complex quantum algorithms, as e.g. Shor's algorithm for quantum factoring [9, 7], can also include classical random numbers, partial measurements, nested evaluation loops and multiple termination conditions; thus the actual quantum operations are embedded into a classical flow-control framework.

In the discussion of non-classical algorithms, both aspects, the classical control structure and the actual quantum operations, are usually treated separately and often, only the latter is formally described [11]. In order to provide a consistent formalism, a quantum programming language will have to resolve this antagonism by generalizing existing classical programming concepts to the field of quantum computing.

1.3 Structured Quantum Programming

In traditional computing science, programming languages can be categorized as either logical (e.g. Prolog), functional (e.g. LISP) or procedural (e.g. Fortran,

Pascal), the latter being the most widely used, for the description of algorithms, as well as the actual implementation of real world programs [1].

Procedural programming languages can be characterized by

- **explicit flow of control**
- **hierarchical program structure**
- **tight mapping between code and computation**

which seems to fit most people's way of reasoning about computational tasks. A procedural language is called *structured*, if flow control is restricted to selection- and loop-statements with well defined entry- and exit-points (e.g. Modula, Pascal without `goto`-statement) [6, 5].

Structured quantum programming is about extending these concepts into the field of quantum computing while preserving their classical semantics. The following table gives an overview of quantum language elements along with their classical counterparts.

classical concept	**quantum analog**
classical machine model	hybrid quantum architecture
variables	quantum registers
subroutines	unitary operators
argument and return types	quantum data types
local variables	scratch registers
dynamic memory	scratch space management
boolean expressions	quantum conditions
conditional execution	conditional operators
selection	quantum if-statement
conditional loops	quantum forking
none	inverse execution of operators
none	quantum measurement

1.4 Hybrid Architecture

Structured quantum programming uses a classical universal language to define the actual sequence of elementary instructions for a quantum computer, so a program is not intended to run on a quantum computer itself, but on a (probabilistic) classical computer, which in turn controls a quantum computer and processes the results of measurements. In the terms of classical computer science, this architecture can be described as a universal computer with a quantum oracle (figure 2).

From the perspective of the user, a quantum program behaves exactly like any other classical program, in the sense that it takes classical input, such as startup parameters or interactive data, and produces classical output.

The state of the controlling computer (i.e. program counter, variable values, but also the mapping of quantum registers) is referred to as *program state*. The

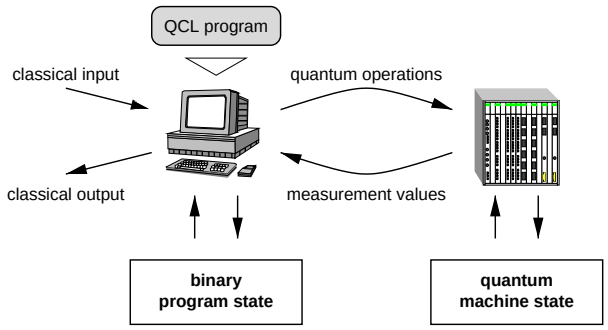

Figure 2: *hybrid quantum architecture*

quantum computer itself does not require any control logic, its computational state can therefore be fully described by the common quantum state $|\Psi\rangle$ of its qubits (*machine state*).

2 The Programming Language QCL

QCL (an acronym for "quantum computation language") is an experimental structured quantum programming language [10]. A QCL interpreter, written in C++, including a numerical simulation library (`libqc`) to emulate the quantum backend is available from

`http://tph.tuwien.ac.at/~oemer/qcl.html`

as free software under the terms of the GPL.

2.1 Quantum Storage and Registers

The smallest unit of quantum storage in QCL is the qubit

Definition 1 (Qubit) *A qubit or quantum bit is a quantum system whose state $|\psi\rangle \in \mathcal{B}$ can be fully described by a superposition of two orthonormal eigenstates labeled $|0\rangle$ and $|1\rangle$, i.e. $\mathcal{B} = \mathbf{C}^2$.*

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \quad \text{with} \quad |\alpha|^2 + |\beta|^2 = 1 \tag{1}$$

QCL treats qubits as quantum registers of length 1.

```
qcl> qureg a[1]; qureg b[1]; // allocate 2 qubits
qcl> Rot(-pi/3,a);           // rotate 1st qubit
qcl> H(b);                   // Hadamard Transformation
```

Definition 2 (Machine State) *The machine state $|\Psi\rangle \in \mathcal{H}$ of an n-qubit quantum computer is the state of a composite system of n identical qubits, i.e. $\mathcal{H} = \mathcal{B}^{\otimes n} = \mathbf{C}^{2^n}$.*

QCL — if used together with a numerical simulator — provides debugging functions which allow the inspection of the otherwise unobservable machine state. In the example below $|\Psi\rangle = |00\rangle \otimes (H|0\rangle) \otimes R_x(\pi/3)|0\rangle$:

```
qcl> dump;                    // show product state as generated above
: STATE: 2 / 4 qubits allocated, 2 / 4 qubits free
0.612372 |0000> + 0.612372 |0010> + 0.353553 |0001> + 0.353553 |0011>
```

Definition 3 (Quantum Register) *An m qubit quantum register* $\mathbf{s}$ *is a sequence of mutually different qubit positions $\langle s_0, s_1 \dots s_{m-1}\rangle$ of some machine state $|\Psi\rangle \in \mathbf{C}^{2^n}$ with $n \geq m$.*

Using an arbitrary permutation π over n elements with $\pi_i = s_i$ for $i < m$, a unitary reordering operator $\Pi_{\mathbf{s}}$ is defined as (see also 2.2.3)

$$\Pi_{\mathbf{s}} |d_0, d_1 \dots d_{n-1}\rangle = |d_{\pi_0}, d_{\pi_1} \dots d_{\pi_{n-1}}\rangle \tag{2}$$

Quantum registers are the fundamental quantum data-type in QCL. As they contain the mapping between the symbolic quantum variables and the actual qubits in the quantum computer, they are the primary interface between the classical frontend and the quantum backend of the hardware architecture, as the machine state can only be accessed via registers.

Quantum registers are dynamically allocated and can be local. Temporary registers can be created by using the qubit- (`q[n]`), subregister- (`q[n:m]`) and concatenation-operators (`q&p`).

2.2 Operators

2.2.1 Register Operators

Definition 4 (Register Operator) *The register operator $U(\mathbf{s})$ for an m-qubit unitary operator $U : \mathbf{C}^{2^m} \to \mathbf{C}^{2^m}$ and an m-qubit quantum register* $\mathbf{s}$ *on an n-qubit quantum computer is the n-qubit operator*

$$U(\mathbf{s}) = \Pi_{\mathbf{s}}^{\dagger} \left(U \otimes I(n-m)\right) \Pi_s \tag{3}$$

with an reordering operator $\Pi_{\mathbf{s}}$ and the k-qubit identity operator $I(k)$.

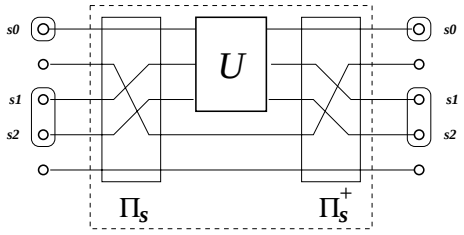

Figure 3: *the register operator $U(\mathbf{s})$*

All operators in QCL are register operators and can also have an arbitrary number of classical parameters. The length of the operand-registers is passed as an additional implicit parameter.

```
operator dft(qureg q) { // Quantum Fourier Transform
  const n=#q;           // set n to length of input
  int i; int j;         // declare loop counters
  for i=1 to n {
    for j=1 to i-1 {    // apply conditional phase gates (see 2.3.2)
      if q[n-i] and q[n-j] { Phase(pi/2^(i-j)); }
    }
    H(q[n-i]);          // qubit rotation
  }
  flip(q);              // swap bit order of the output
}
```

QCL operators are unitary and have mathematical semantics i.e. their effect must be reproduceable and may only depend on the specified parameters. This esp. excludes

- dependencies on the program state (e.g. global variables)
- side effects on the program state
- user input and calls to `random()`
- non-unitary quantum operations (i.e. calls to `measure` and `reset`)

For any QCL operator, the adjoint operator is determined on the fly if the call is prefixed with the inversion-flag (`!`).[1]

```
qcl> qureg q[2];             // allocate a 2-qubit register
qcl> dft(q);                 // discrete Fourier transform
[2/32] 0.5 |00> + 0.5 |01> + 0.5 |10> + 0.5 |11>
qcl> !dft(q);                // inverse transform
[2/32] 1 |00>
```

2.2.2 Quantum Data Types

Classical programming languages often allow to impose access restrictions on variables and subroutine parameters. This is equally done to prevent subsequent programming errors, as to provide information to the compiler to allow for more efficient optimizations.

QCL extends this concept to quantum registers by introducing quantum data types to limit the ways how operators may effect the machine state.

type	**restriction**
`qureg`	none
`quconst`	invariant to all suboperators
`quvoid`	has to be empty when the uninverted operator is called
`quscratch`	has to be empty before and after the call

[1] Internally, this is achieved by recursively caching all suboperator calls and executing them in reverse order with swapped inversion-flags.

Definition 5 (Invariance of Registers) *A quantum register* $\mathbf{c}$ *is invariant to a register operator* $U(\mathbf{s}, \mathbf{c})$ *iff* $U\,|i\rangle_{\mathbf{s}}|j\rangle_{\mathbf{c}} = (U_j\,|i\rangle_{\mathbf{s}})\;|j\rangle_{\mathbf{c}}$ *with unitary* U_j*.*

Definition 6 (Empty Registers) *A register* $\mathbf{e}$ *is empty iff* $|\Psi\rangle = |0\rangle_{\mathbf{e}}|\psi'\rangle$

2.2.3 Quantum Functions

One important aspect of quantum computing is, that — due to the linearity of unitary transformations — an operator applied to a superposition state $|\Psi\rangle$ is simultaneously applied to all basis vectors that constitute $|\Psi\rangle$ (*quantum parallelism*) since

$$U \sum_i c_i|i\rangle = \sum_i c_i(U\,|i\rangle) \tag{4}$$

In many cases U implements a reversible boolean, or, equivalently, a bijective integer function, by treating the basis vectors merely as bitstrings or binary numbers.

Definition 7 *A n-qubit quantum function is a unitary operator of the form* $U : |i\rangle \to |\pi_i\rangle$ *with some permutation* π *over* $\mathbf{Z}_{2^n}$*.*

Quantum functions are implemented by the QCL subroutine-type `qufunct`.

```
qufunct inc(qureg x) {      // increment register
  int i;
  for i = #x-1 to 1 step -1 {
    CNot(x[i],x[0:i-1]);    // apply controlled-not from
  }                         // MSB to LSB
  Not(x[0]);
}
```

To enforce the above restrictions, the 4 QCL subroutines types form a calling hierarchy, i.e. routines may only invoke subroutines of the same or a lower level.

subroutine	S	Ψ	**invertible**	**description**
`procedure`	all	all	no	classical control structure
`operator`	none	unitary	yes	general unitary operator
`qufunct`	none	permutation	yes	quantum function
functions	none	none	no	mathematical functions

The columns S and ψ denote the allowed side-effects on the classical program state and the quantum machine state.

2.2.4 Irreversible Functions

One obvious problem in QC is its restriction to reversible computations. Consider a simple operation like computing the parity of a bitstring

$$\texttt{parity}' : |i\rangle \to |b(i) \bmod 2\rangle \quad \text{with} \quad b(n) = n \bmod 2 + b(\lfloor n/2 \rfloor),\, b(0) = 0 \tag{5}$$

Clearly, this operation is non-reversible since $\texttt{parity}'\,|1\rangle = \texttt{parity}'\,|2\rangle$, so `parity'` is not an unitary operator. However, if we use an additional target register, then we can define an operator `parity` which matches the condition $\texttt{parity}\,|i, 0\rangle = |i, b(i) \bmod 2\rangle$.

```
qufunct parity(quconst x,quvoid y) {
  int i;
  for i = 0 to #x-1 {
    CNot(y,x[i]);           // flip parity for each set bit
  }
}
```

In QCL, an operator $F : |x\rangle_{\mathbf{x}}|0\rangle_{\mathbf{y}} \rightarrow |x\rangle_{\mathbf{x}}|f(x)\rangle_{\mathbf{y}}$ is declared as `qufunct` with at least one invariant (`quconst`) argument register $\mathbf{x}$ and one empty (`quvoid`) target register $\mathbf{y}$ as parameter. The result for $F|x\rangle_{\mathbf{x}}|y \neq 0\rangle_{\mathbf{y}}$ is unspecified to allow for different ways to accumulate the result. So $F' : |x, y\rangle \rightarrow |x, y \oplus f(x)\rangle$ and $F'' : |x, y\rangle \rightarrow |x + y \bmod 2^k\rangle$ are merely considered to be different implementations of the same quantum function.

2.2.5 Scratch Space Management

While quantum functions can be used to work around the reversible nature of QC, the necessity to keep a copy of the argument is a problem, as longer computations will leave registers filled with intermediate results.

Let F be a quantum function with the argument register $\mathbf{x}$ (`quconst`), the target register $\mathbf{y}$ (`quvoid`) and the scratch register $\mathbf{s}$ (`quscratch`)

$$F(\mathbf{x}, \mathbf{y}, \mathbf{s}) : |i\rangle_{\mathbf{x}}|0\rangle_{\mathbf{y}}|0\rangle_{\mathbf{s}} \rightarrow |i\rangle_{\mathbf{x}}|f(i)\rangle_{\mathbf{y}}|j(i)\rangle_{\mathbf{s}} \tag{6}$$

F fills the register $\mathbf{s}$ with the temporary junk bits $j(i)$. To reclaim $\mathbf{s}$, QCL transparently allocates an auxiliary register $\mathbf{t}$ and translates F into an operator F' which is defined as [3]

$$F'(\mathbf{x}, \mathbf{y}, \mathbf{s}, \mathbf{t}) = F^{\dagger}(\mathbf{x}, \mathbf{t}, \mathbf{s})\, \mathtt{fanout}(\mathbf{t}, \mathbf{y})\, F(\mathbf{x}, \mathbf{t}, \mathbf{s}) \tag{7}$$

The *fanout* operator is a quantum function defined as

$$\mathtt{fanout} : |i\rangle|0\rangle \rightarrow |i\rangle|i\rangle \tag{8}$$

The application of F' restores the scratch register $\mathbf{s}$ and the auxiliary register $\mathbf{t}$ to $|0\rangle$ while preserving the function value in the target register $\mathbf{y}$:

$$|i, 0, 0, 0\rangle \rightarrow |i, 0, j(i), f(i)\rangle \rightarrow |i, f(i), j(i), f(i)\rangle \rightarrow |i, f(i), 0, 0\rangle \tag{9}$$

2.3 Quantum Conditions

2.3.1 Conditional Operators

Classical programs allow the conditional execution of code in dependence on the content of a boolean variable (conditional branching).

A unitary operator, on the other hand, is static and has no internal flow-control. Nevertheless, we can conditionally apply an n-qubit operator U to a quantum register by using an *enable* qubit and define an $n + 1$ qubit operator U'

$$U' = \begin{pmatrix} I(n) & 0 \\ 0 & U \end{pmatrix} \tag{10}$$

So U is only applied to base-vectors where the enable bit is set. This can be easily extended to enable-registers of arbitrary length.

Definition 8 *A conditional operator $U_{[[\mathbf{e}]]}$ with the enable register* $\mathbf{e}$ *is a unitary operator of the form*

$$U_{[[\mathbf{e}]]} : |i,\epsilon\rangle = |i\rangle|\epsilon\rangle_{\mathbf{e}} \rightarrow \begin{cases} (U|i\rangle)\,|\epsilon\rangle_{\mathbf{e}} & \text{if } \epsilon = 111\ldots \\ |i\rangle|\epsilon\rangle_{\mathbf{e}} & \text{otherwise} \end{cases}$$

A conditional version of the increment operator from 2.2.3 can be explicitly implemented as

```
qufunct cinc(qureg x,quconst e) {
  int i;
  for i = #x-1 to 1 step -1 { CNot(x[i],x[0:i-1] & e); }
  CNot(x[0],e);
}
```

QCL can automatically derive $U_{[[\mathbf{e}]]}$ for an operator U if its declaration is prefixed with `cond`.

```
cond qufunct inc(qureg x,quconst e) { ... }
```

The enable register can be set by a quantum `if`-statement and QCL will transparently transform the defined operator into its conditional version when necessary.

2.3.2 Quantum if-statement

Just like the concept of quantum functions allows the computation of irreversible boolean functions with unitary operators, conditional operators allow conditional branching depending on unobservable qubits.

Given the above definitions, the statement `if e { inc(x); }` is equivalent to the explicit call of `cinc(x,e)` and `if e { inc(x); } else { !inc(x); }` is equivalent to the sequence

```
cinc(x,e);      // conditional increment
Not(e);         // invert enable qubit
!cinc(x,e);     // conditional decrement
Not(e);         // restore enable qubit
```

Quantum if-statements can be nested. Since operators within the if- and else-branches are transformed into their conditional versions, they must be declared `cond` and must not operate on any qubits used in the condition.

2.3.3 Complex Conditions

Conditions in quantum if-statements are not restricted to single qubits, but can contain any boolean expression and also allow the mixing of classical and quantum bits.

```
qcl> qureg q[4]; qureg b[1]; qureg a[1];
qcl> H(a & b);  // prepare test-state
[6/32] 0.5 |000000> + 0.5 |010000> + 0.5 |100000> + 0.5 |110000>
qcl> if a and b { inc(q); }
[6/32] 0.5 |000000> + 0.5 |010000> + 0.5 |100000> + 0.5 |110001>
qcl> if a or b { inc(q); }
[6/32] 0.5 |000000> + 0.5 |010001> + 0.5 |100001> + 0.5 |110010>
qcl> if not (a or b) { inc(q); }
[6/32] 0.5 |000001> + 0.5 |010001> + 0.5 |100001> + 0.5 |110010>
```

QCL produces a sequence of `CNot`-gates to evaluate a *quantum condition*.[2] If necessary, scratch qubits are transparently allocated and uncomputed again.

2.3.4 Forking if-statement

If the body of a quantum if-statement contains statements which change the program state (e.g. assignments to local variables), then subsequent operator calls may differ, depending on whether the if- or the else-branch has been executed.

In that case, QCL follows all possible classical paths throughout the operator definition (forking), accumulates the conditions of all visited quantum if-statements and serializes the generated sequence of operators.

```
cond qufunct demux(quconst s,qureg q) {
  int i;
  int n = 0;
  for i=0 to #s-1 {             // accumulate content of
    if s[i] { n=n+2^i; }        //   selection register in a
  }                             //   classical variable
  Not(q[n]);                    // flip selected output qubit
}
```

Figure 4 shows the quantum circuit [8] generated by `demux(s,q)` in the case of a 2-qubit selection register **s**.

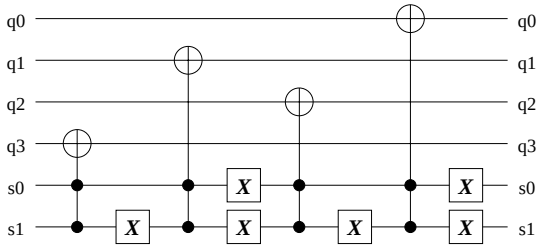

Figure 4: *a quantum demultiplexer*

Forking if-statements may only appear within an operator definition to assure that the different execution threads can be joined again.

[2] Internally, this is achieved by transforming a quantum condition into its exclusive disjunctive normal form [2].

3 Conclusions

Throughout the history of classical computing, hardware development has always been accompanied by a corresponding improvement of programming methodology. The formalism of quantum circuits — the moral equivalent to hand written machine code — seems inadequate for quantum computers with more than a couple of qubits, so more abstract methods will eventually be required.

We have demonstrated how well established concepts of classical programming languages like subroutines, local variables or conditional branching can be ported to the field of quantum computing. Besides providing a new level of abstraction, we also hope that a quantum programming language which semantically integrates those concepts will allow for a better and more intuitive understanding of non-classical algorithms.

References

[1] Aaby, A.A. *1996 Introduction to Programming Languages.* `http://cs.wwc.edu/~aabyan/221_2/PLBOOK/`

[2] Bani-Eqbal, B. *1992 Exclusive Normal Form of Boolean Circuits. Technical Report Series, UMCS-92-4-1, University of Manchester,* `http://www.cs.man.ac.uk/cstechrep/Abstracts/UMCS-92-4-1.html`

[3] Bennett, C.H., "Logical Reversibility of Computation" in *IBM J. Res. Develop.* **17**, p. 525 (1973)

[4] Deutsch, D., "Quantum Computational Networks" in *Proceedings of the Royal Society London*, **A 439**, 1989, pp. 553-558.

[5] Dahl, O.-J., Dijkstra, E.W., and Hoare, C.A.R. *1972 Structured Programming, Academic Press, London, UK (1972)*

[6] E.W. Dijkstra *1969 Structured Programming, Software Engineering Techniques, Buxton, J. N., and Randell, B., eds. Brussels, Belgium, NATO Science Committee.*

[7] Ekert, A. and Jozsa, R., "Shor's Quantum Algorithm for Factoring Numbers" in *Rev. Modern Physics* **68(3)**, pp. 733-753 (1996)

[8] Nielsen, M.A. and Chuang, I.L. *2000 Quantum Computation and Quantum Information, Cambridge University Press*

[9] Shor, P.W., "Algorithms for Quantum Computation: Discrete Logarithms and Factoring" in *Proceedings of the 35th IEEE Symposium on Foundations of Computer Science*, S. Goldwasser (Ed.), IEEE Computer, Society Press, 1994, pp. 124-134.

[10] Ömer, B., *A Procedural Formalism for Quantum Computing, master-thesis, Technical University Vienna, 1998.* `http://tph.tuwien.ac.at/~oemer/qcl.html`

[11] Wallace, J., "Quantum Computer Simulators" in *Partial Proceedings of the 4th Int. Conference CASYS 2000*, D. M. Dubois (Ed.), International Journal of Computing Anticipatory Systems, volume 10, 2001, pp. 230-245.